Parallel Programming with Compiler Directives — OpenMP —

Clemens Grelck
University of Amsterdam

ASCI Course A24
A Programmer’s Guide for Modern High-Performance Computing Architectures
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions
Design Rationale of OpenMP

**Ideal:**

- Automatic parallelisation of sequential code.
- No additional parallelisation effort for development, debugging, maintenance, etc.

**Problem:**

- Data dependences are difficult to assess.
- Compilers must be conservative in their assumptions.

**Way out:**

- Take or write ordinary sequential program.
- Add annotations/pragmas/compiler directives that guide parallelisation.
- Let the compiler generate the corresponding code.
Design Rationale of OpenMP

**Ideal:**
- Automatic parallelisation of sequential code.
- No additional parallelisation effort for development, debugging, maintenance, etc.

**Problem:**
- Data dependences are difficult to assess.
- Compilers must be conservative in their assumptions.
Design Rationale of OpenMP

Ideal:

▶ Automatic parallelisation of sequential code.
▶ No additional parallelisation effort for development, debugging, maintenance, etc.

Problem:

▶ Data dependences are difficult to assess.
▶ Compilers must be conservative in their assumptions.

Way out:

▶ Take or write ordinary sequential program.
▶ Add annotations/pragmas/compiler directives that guide parallelisation.
▶ Let the compiler generate the corresponding code.
Short History of OpenMP

Towards OpenMP:

- Mid 1980s: First commercial SMPs appear.
  - ANSI standard X3H5 (not adopted)
- Late 1980s to early 1990s:
  - Distributed memory architectures prevail.
- Early 1990s until today:
  - Shared memory architectures significantly gain popularity.
  - Different vendor-specific compiler extensions.
- 1996/97:
  - Development of a new industry standard: OpenMP
  - Actively supported by all major hardware vendors: Intel, HP, SGI, IBM, SUN, Compaq, ...
  - OpenMP Architectural Review Board:
    http://www.openmp.org/
Short History of OpenMP

OpenMP Standardisation:

▶ 1996 : Fortran Version 1.0
▶ 1998 : C/C++ Version 1.0
▶ 1999 : Fortran Version 1.1
▶ 2000 : Fortran Version 2.0
▶ 2002 : C/C++ Version 2.0
▶ 2005 : Joint Version 2.5
▶ 2008 : Joint Version 3.0
▶ 2011 : Joint Version 3.1
OpenMP Architectural Model

**Characteristics:**

- Multiple processors
- Multiple cores
- Shared memory (with cache coherence)

![Diagram of OpenMP Architectural Model]

Clemens Grelck, University of Amsterdam
Potential Target Architectures

- Intel/AMD standard multicores
- Symmetric multiprocessors
- Oracle Niagara series
OpenMP at a Glance

OpenMP as a programming interface:

- Compiler directives
- Library functions
- Environment variables

C/C++ version:

```cpp
#pragma omp name [clause]*
structured block
```

Fortran version:

```fortran
!$ OMP name [ clause [, clause]*]
code block
!$ OMP END name
```
Hello World with OpenMP

```c
#include "omp.h"
#include <stdio.h>

int main()
{
    printf( "Starting execution with %d threads:\n", 
            omp_get_num_threads());

    #pragma omp parallel
    {
        printf( "Hello world says thread %d of %d.\n", 
                omp_get_thread_num(), 
                omp_get_num_threads());
    }

    printf( "Execution of %d threads terminated.\n", 
            omp_get_num_threads());

    return( 0);
}
```
Hello World with OpenMP

Compilation:

gcc -fopenmp hello_world.c

Output using 4 threads:

Starting execution with 1 threads:
Hello world says thread 2 of 4.
Hello world says thread 3 of 4.
Hello world says thread 1 of 4.
Hello world says thread 0 of 4.
Execution of 1 threads terminated.
Hello World with OpenMP

Using 4 threads:

Starting execution with 1 threads:
Hello world says thread 2 of 4.
Hello world says thread 3 of 4.
Hello world says thread 1 of 4.
Hello world says thread 0 of 4.
Execution of 1 threads terminated.

Who determines number of threads ?

- Environment variable: export OMP_NUM_THREADS=4
- Library function: void omp_set_num_threads( int)
OpenMP Execution Model

Classical Fork/Join:

- Master thread executes serial code.
- Master thread encounters parallel directive.
- Master and slave threads concurrently execute parallel block.
- Implicit barrier, wait for all threads to finish.
- Master thread resumes serial execution.
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions
Example: element-wise vector product:

```c
void elem_prod( double *c, double *a, double *b, int len)
{
    int i;

    #pragma omp parallel for
    for (i=0; i<len; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```
Simple Loop Parallelisation

Example: element-wise vector product:

```c
void elem_prod( double *c, double *a, double *b, int len)
{
    int i;

    #pragma omp parallel for
    for (i=0; i<len; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```

Prerequisite:

- No data dependence between any two iterations.
Simple Loop Parallelisation

Example: element-wise vector product:

```c
void elem_prod( double *c, double *a, double *b, int len )
{
    int i;

    #pragma omp parallel for
    for (i =0; i< len ; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```

Prerequisite:

- No data dependence between any two iterations.
- Caution: YOU claim this property !!
Directive `#pragma omp parallel for`

What the compiler directive does for you:

- It starts additional worker threads depending on `OMP_NUM_THREADS`.
- It divides the iteration space among all threads.
- It lets all threads execute loop restricted to their mutually disjoint subsets.
- It synchronizes all threads at an implicit barrier.
- It terminates worker threads.
Directive `#pragma omp parallel for`

**What the compiler directive does for you:**

- It starts additional worker threads depending on `OMP_NUM_THREADS`.
- It divides the iteration space among all threads.
- It lets all threads execute loop restricted to their mutually disjoint subsets.
- It synchronizes all threads at an implicit barrier.
- It terminates worker threads.

**Restrictions:**

- The directive must directly precede for-loop.
- The for-loop must match a constrained pattern.
- The trip-count of the for-loop must be known in advance.
Shared and Private Variables

Example:

```c
#pragma omp parallel for
for (i=0; i<len; i++)
{
    res[i] = a[i] * b[i];
}
```

- Shared variable: one instance for all threads
- Private variable: one instance for each thread
Shared and Private Variables

Example:

```c
#pragma omp parallel for
for (i = 0; i < len; i++)
{
    res[i] = a[i] * b[i];
}
```

Who decides that res, a, b, and len are shared variables, whereas i is private??
Example:

```c
#pragma omp parallel for
for (i = 0; i < len; i++)
{
    res[i] = a[i] * b[i];
}
```

Who decides that res, a, b, and len are shared variables, whereas i is private??

Default rules:

- All variables are **shared**.
- Only loop variables of parallel loops are **private**.
Parallelisation of a Less Simple Loop

**Mandelbrot set:**

double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
Parallelisation of a Less Simple Loop

Mandelbrot set:

double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}

Properties to check:

▶ No data dependencies between loop iterations?
Parallelisation of a Less Simple Loop

**Mandelbrot set:**

```c
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
```

**Properties to check:**

- No data dependencies between loop iterations? **YES!**
- Trip-count known in advance? **YES!**
Parallelisation of a Less Simple Loop

**Mandelbrot set:**

```c
double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}
```

**Properties to check:**

- No data dependencies between loop iterations? **YES!**
- Trip-count known in advance? **YES!**
- Function `mandelval` without side-effects? [ ]

Clemens Grelck, University of Amsterdam

Compiler Directives: OpenMP
Parallelisation of a Less Simple Loop

**Mandelbrot set:**

double x, y;
int i, j, max = 200;
int depth[M,N];

... for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
} }

**Properties to check:**

- No data dependencies between loop iterations? **YES!**
- Trip-count known in advance? **YES!**
- Function `mandelval` without side-effects? **YES!**
- Only loop variable `i` needs to be private? **NO!!!**

Clemens Grelck, University of Amsterdam

Compiler Directives: OpenMP
Parallelisation of a Less Simple Loop

Mandelbrot set:

double x, y;
int i, j, max = 200;
int depth[M,N];
...
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    x = (double) i / (double) M;
    y = (double) j / (double) N;
    depth[i,j] = mandelval( x, y, max);
  }
}

Properties to check:

▶ No data dependencies between loop iterations? YES!
▶ Trip-count known in advance? YES!
▶ Function mandelval without side-effects? YES!
▶ Only loop variable i needs to be private? NO !!!!

Check x,y,j
Parallelisation of a Less Simple Loop

**Mandelbrot set:**

```c
double x, y;
int i, j, max = 200;
int depth[M,N];
...
#pragma omp parallel for private(x, y, j)
for (i =0; i<M; i++) {
    for (j =0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval(x, y, max);
    }
}
```

**Private clause:**

- Directives may be refined by *clauses*.
- Private clause allows to tag any variable as private.
- **Caution**: private variables are **not** initialised outside parallel section !!!
Parallelisation of a Less, Less Simple Loop

Mandelbrot set with additional counter:

```c
int total = 0;
...
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i, j] = mandelval(x, y, max);
        total = total + depth[i, j];
    }
}
```

Problems:
- New variable `total` introduces data dependence.
- Data dependence could be ignored due to associativity.
- New variable `total` must be shared.
- Incrementation of `total` must avoid race condition.
Mandelbrot set with additional counter:

```c
int total = 0;
...
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
        total = total + depth[i,j];
    }
}
```

Problems:

- New variable `total` introduces data dependence.
- Data dependence could be ignored due to associativity.
- New variable `total` must be shared.
- Incrementation of `total` must avoid race condition.
Mandelbrot set with additional counter:

```c
int total = 0;
...
#pragma omp parallel for private(x, y, j)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval(x, y, max);

        #pragma omp critical
        {
            total = total + depth[i,j];
        }
    }
}
```
Critical Regions

The critical directive:

- Directive must immediately precede new statement block.
- Statement block is executed without interleaving.
- Directive implements critical region.

Equivalence:

```c
#pragma omp critical
{
  <statements>
}
```

```c
pthread_mutex_lock( &lock);
<statements>
pthread_mutex_unlock( &lock);
```
Critical Regions

**The critical directive:**

- Directive must immediately precede new statement block.
- Statement block is executed without interleaving.
- Directive implements critical region.

**Equivalence:**

```
#pragma omp critical
{
    <statements>
}
```

```
pthread_mutex_lock( &lock);
<statements>
pthread_mutex_unlock( &lock);
```

**Disadvantage:**

- All critical regions in entire program are synchronised.
- Unnecessary overhead.
### Critical Regions

The named critical directive

- Critical regions may be associated with names.
- Critical regions with identical names are synchronised.
- Critical regions with different names are executed concurrently.

**Equivalence:**

```c
#pragma omp critical (name)
{
    <statements>
}

pthread_mutex_lock( &name_lock);
<statements>
pthread_mutex_unlock( &name_lock);
```
Reduction Operations

**Specific solution: reduction clause**

```c
#pragma omp parallel for private(x, y, i, j) reduction(+: total)
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i, j] = mandelval(x, y, max);
        total = total + depth[i, j];
    }
}
```

**Properties:**

- Reduction clause only supports built-in reduction operations: +, *, ^, &, |, &&, ||, min, max.
- User-defined reductions only supported via critical regions.
- Bit accuracy not guaranteed.
Shared and Private Variables Reloaded

**Shared variables:**
- One instance shared between sequential and parallel execution.
- Value unaffected by transition.

**Private variables:**
- One instance during sequential execution.
- One instance per worker thread during parallel execution.
- No exchange of values.

New: Firstprivate variables:
- Like private variables, but...
- Worker thread instances initialised with master thread value.

New: Lastprivate variables:
- Like private variables, but...
- Master thread instance updated to value of worker thread instance that executes the last (in sequential order) iteration.
Shared and Private Variables Reloaded

**Shared variables:**
- One instance shared between sequential and parallel execution.
- Value unaffected by transition.

**Private variables:**
- One instance during sequential execution.
- One instance per worker thread during parallel execution.
- No exchange of values.

**New: Firstprivate variables:**
- Like private variables, but ...
- Worker thread instances initialised with master thread value.
Shared and Private Variables Reloaded

**Shared variables:**
- One instance shared between sequential and parallel execution.
- Value unaffected by transition.

**Private variables:**
- One instance during sequential execution.
- One instance per worker thread during parallel execution.
- No exchange of values.

**New: Firstprivate variables:**
- Like private variables, but ...
- Worker thread instances initialised with master thread value.

**New: Lastprivate variables:**
- Like private variables, but ...
- Master thread instance updated to value of worker thread instance that executes the last (in sequential order) iteration.
Example:

```c
int a=1, b=2, c=3, d=4, e=5;

#pragma omp parallel for private(a) 
     firstprivate(b,d) 
     lastprivate(c,d)
for (i=0; i<10; i++) {
    // before first iteration:
    // a : ?? | b : ?? | c : ?? | d : ?? | e: ??
    a++; b++; c=7; d++;
}

// a : ?? | b : ?? | c : ?? | d : ?? | e: ??
```

Clemens Grelck, University of Amsterdam
Compiler Directives: OpenMP
Example:

```c
int a=1, b=2, c=3, d=4, e=5;

#pragma omp parallel for private(a) 
    firstprivate(b,d) 
    lastprivate(c,d)
for (i=0; i<10; i++) {
    // before first iteration:
    // a : undef | b : 2 | c : undef | d : 4 | e : 5
    a++; b++; c=7; d++;
}

// a : 1 | b : 2 | c : 7 | d : ?? | e : 5
```

Note:

- The value of d depends on the number of iterations executed by the thread that is assigned iteration i=9 by the OpenMP loop scheduler.
Conditional Parallelisation

**Problem:**

- Parallel execution of a loop incurs overhead:
  - creation of worker threads
  - scheduling
  - synchronisation barrier
- This overhead must be outweighed by sufficient workload.
- Workload depends on
  - loop body,
  - trip count.

Clemens Grelck, University of Amsterdam
Compiler Directives: OpenMP
Conditional Parallelisation

Problem:
- Parallel execution of a loop incurs overhead:
  - creation of worker threads
  - scheduling
  - synchronisation barrier
- This overhead must be outweighed by sufficient workload.
- Workload depends on
  - loop body,
  - trip count.

Example:
```c
if (len < 1000) {
    for (i=0; i<len; i++)
        { 
            res[i] = a[i] * b[i];
        }
}
else {
    #pragma omp parallel for
    for (i=0; i<len; i++)
        { 
            res[i] = a[i] * b[i];
        }
}
```
### Conditional Parallelisation

**Introducing the if-clause:**

```c
if (len < 1000) {
    for (i=0; i<len; i++) {
        res[i] = a[i] * b[i];
    }
}
else {
    #pragma omp parallel for
    for (i=0; i<len; i++) {
        res[i] = a[i] * b[i];
    }
}

#pragma omp parallel for if (len >= 1000)
for (i=0; i<len; i++) {
    res[i] = a[i] * b[i];
}
```
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions
Loop Scheduling

Definition:

- Loop scheduling determines which iterations are executed by which thread.

Aim:

- Equal workload distribution
Loop Scheduling

Problem:
- Different situations require different techniques

The `schedule` clause:

```c
#pragma omp parallel for schedule( <type> [, <chunk>])
for (...) {
    ...
}
```

Properties:
- Clause selects one out of a set of scheduling techniques.
- Optionally, a chunk size can be specified.
- Default chunk size depends on scheduling technique.
Loop Scheduling

**Static scheduling:**

```c
#pragma omp parallel for schedule(static)
```

- Loop is subdivided into as many chunks as threads exist.
- Often called **block scheduling**.
Loop Scheduling

Static scheduling:

```
#pragma omp parallel for schedule(static)
```

- Loop is subdivided into as many chunks as threads exist.
- Often called block scheduling.

Static scheduling with chunk size:

```
#pragma omp parallel for schedule(static, <n>)
```

- Loop is subdivided into chunks of $n$ iterations.
- Chunks are assigned to threads in a round-robin fashion.
- Also called block-cyclic scheduling.
Loop Scheduling

**Dynamic scheduling:**

```c
#pragma omp parallel for schedule( dynamic, <n>)
```

- Loop is subdivided into chunks of \( n \) iterations.
- Chunks are dynamically assigned to threads on their demand.
- Also called **self scheduling**.
- Default chunk size: 1 iteration.

**Properties:**

- Allows for dynamic load distribution and adjustment.
- Requires additional synchronization.
- Generates more overhead than static scheduling.
Loop Scheduling

Dilemma of chunk size selection:

- Small chunk sizes mean good load balancing, but high synchronisation overhead.
- Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.
Dilemma of chunk size selection:

- Small chunk sizes mean good load balancing, but high synchronisation overhead.
- Large chunk sizes reduce synchronisation overhead, but result in poor load balancing.

Rationale of guided scheduling:

- In the beginning, large chunks keep synchronisation overhead small.
- When approaching the final barrier, small chunks balance workload.
Loop Scheduling

**Guided scheduling:**

```
#pragma omp parallel for schedule( guided, <n>)
```

- Chunks are dynamically assigned to threads on their demand.
- Initial chunk size is implementation dependent.
- Chunk size decreases exponentially with every assignment.
- Also called **guided self scheduling**.
- Minimum chunk size: \( n \) (default: 1)

**Example:**

- Total number of iterations: 250
- Initial / minimal chunk size: 50 / 5
- Current chunk size: 80% of last chunk size:
Loop Scheduling

Can’t decide? Use runtime scheduling:

```c
#pragma omp parallel for schedule(runtime)
```

▶ Choose scheduling at runtime
▶ Environment variable `OMP_SCHEDULE`
▶ Library function
  ```c
  void omp_set_schedule(omp_sched_t kind, int modifier)
  ```

For the records:

```c
typedef enum omp_sched_t {
  omp_sched_static = 1,
  omp_sched_dynamic = 2,
  omp_sched_guided = 3,
  omp_sched_auto = 4
} omp_sched_t;
```
Mandelbrot Reloaded

Previous code:

double x, y;
int i, j, max = 200;
int depth[M,N];
...
#pragma omp parallel for private(i, j, x, y)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval(x, y, max);
    }
}

Which scheduling?
With guided scheduling:

double x, y;
int i, j, max = 200;
int depth[M,N];
...
#pragma omp parallel for private( i, j, x, y) scheduling( guided)
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval( x, y, max);
    }
}

What if M is small?
Mandelbrot Reloaded

With collapse directive:

double x, y;
int i, j, max = 200;
int depth[M,N];
...
# pragma omp parallel for private(i, j, x, y) \
   scheduling(guided) \
   collapse(2)

for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        x = (double) i / (double) M;
        y = (double) j / (double) N;
        depth[i,j] = mandelval(x, y, max);
    }
}

Collapse directive:

- Collapses multi-dimensional iteration space into single one subject to OpenMP scheduling
- Requires perfect loop nest
- Advantage: more control/flexibility for OpenMP
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions
Parallel Programming with Compiler Directives: OpenMP

OpenMP at a Glance

Loop Parallelization

Scheduling

Parallel Code Regions

Advanced Concepts

Conclusions